NORWEGIAN UNIVERSITY OF
SCIENCE AND TECHNOLOGY

TDT4305 - BIG DATA ARCHITECHTURE

Project - Part 1

Authors

Gaute HÅHJEM SOLEMDAL

Håkon STRANDLIE

Spring 2020

Contents

1 Introduction **2**

2 Tasks **3**

2.1 Task 1 3

2.2 Task 2 3

2.3 Task 3 5

2.4 Task 4 5

2.5 Task 5 6

2.6 Task 6 6

1 Introduction

This document will briefly explain the solution for each task and subtask as deliverables for part 1 of the project in the NTNU course Big Data Architectures. The source code for the solution is not provided in this document, but is delivered together with this document, and can be built and run by following the README in the code repository. The code is written in Scala and built using sbt.

2 Tasks

2.1 Task 1

a - Load the dataset into separate RDDs and count the number of the rows in each RDD

This simple task is solved by using the `.textFile(path)` on the SparkContext that is availiable. The number of rows are printed using the `.count` method on each RDD.

2.2 Task 2

a - How many distinct users are there in the dataset?

For this task, three operations are performed.

1. `.map(review => review(1)`
    - Since the only relevant column is user_id, we do a projection to remove all columns that are not relevant.
2. `.filter(id => id != """"user_id"""")`
    - Remove the header row
3. `.map (id => if (id(0).toString == """""""") id.slice(1, id.length - 1) else id)`
    - Since some id's are stored with quotes (") before and after the actual id, we remove the quotes from the id's that have a quote as the first character.

The number of results from these transformations are counted and outputted.

b - How many what is the average number of the characters in a user review?

Three operations are performed first.

1. `.map(review => review(3)`
    - Since the only relevant column is review_text, we do a projection to remove all columns that are not relevant.
2. `.filter( text => ... )`
    - Remove the header row, and all rows with illegal base64-encoded characters
3. `.map(base64_text => {new String(java.util.Base64.getDecoder.decode(base64_text))}`
    - Decode the base64 encoded text into a human readable string.

At this point we have an RDD called `review_texts` with an array of Java Strings, and we call the `.length` method of each text in `review_texts` to get the sum of characters. After this we sum and divide by the number of reviews to find the average.

c - What is the business_id of the top 10 businesses with the most number of reviews

This task is solved by first removing the header row and then removing quotes from the id if it exists. Then the number of reviews are counted for each business_id, they are sorted according to this count and the top 10 results are output.

d - Find the number of reviews per year

Since each date is stored as a Unix timestamp in the file, we use Java's SimpleDateFormat to convert to a year. Using this we map each timestamp to a year, count the rows corresponding to a year, and output the list.

Note: We had to use take() to get the list sorted correctly. Using only foreach(println) did not produce the correct result.

e - What is the time and date of the first and last review?

Here we took each review date, and using the SimpleDateFormat from java.text (same as above) we formatted the `min_date` found by `.reduce((a, b) => Math.min(a, b)` and `max_date` found correspondingly. These are then output.

f - Calculate the Pearson correlation coefficient (PCC)

To solve this task we took a step-by-step approach.

1. Remove irrelevant columns, clean up the data, decode the text and group the reviews by the user_id
2. Calculate the **number of reviews** and **average number of characters** per user_id
3. Calculate the **average number of reviews** and **average number of characters** across the dataset
4. Observe that the formula for the PCC consists of

   $$X_i - \bar{x}$$

   and

   $$Y_i - \bar{Y}$$

   in two combinations. I.e an efficient solution is to calculate the numerical value of these once and re-use them. We do this in the `differences` RDD
5. The final value for PCC is calculated using the `differences` RDD and multiplications on this.

2.3 Task 3

a - What is the average rating for businesses in each city?

We solved this task by first making two key/value RDDs; (1) city with a count of how many individual ratings belonging to each city - using `.countByKey`, and (2) city with the accumulated score, i.e. sum of star ratings, belonging to each city - using `.reduceByKey`. With the accumulated score and count we calculated the average rating in a `.map`.

b - What are the top 10 most frequent categories in the data?

Since the category field was a string potentially containing several categories we had to "flatten" this field into several rows. We did this using `.flatMap`. Then it was just to count the number of occurences of each category with a `.reduceByKey`, after mapping each row into a tuple like `(category, 1)` such that it adds 1 correctly for each row. We then sorted by the accumulated score in descending order in a `.sortBy`, to prepare for the final `.take(10)`.

c - Calculate the geographical centroid of the region shown by the postal code

We solved this task in essentially the same manner as task a; to get the average latitude and longitude we need to divide the sum of items by the number of items. The difference is that in this task we have two values, instead of one, for each row. To work around this problem we used `.map` to create a new key/value RDD where they value was a 2-tuple, containing the latitue and longitude values: `(postcode, (latitude,longitude))`. Thus, we had to use the 'tuple syntax' to reach the correct values of the tuple such as: `.reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))`

2.4 Task 4

a - Find the top 10 nodes with the most number of in and out degrees in the friendship graph

This solution begins by cleaning up the elements and removing quotes from the id's. Then we create key-value RDDs with id's as key and a list of id's as value, and simply sort the RDD according to the length of the list and output the top 10.

b - Find the mean and median for number of in and out degrees in the friendship graph

Here we also do some initial cleanup, before we create key-value RDDs, as above, with id's as key and a list of id's as value for both in and out edges and sum these numbers and calculate the averages. We also calculate the midpoint and use it with the lookup-function to find the medians. Both averages and medians are output.

2.5 Task 5

a - Load each file in the dataset into separate Spark's DataFrames

Here we first define a schema for *review_table*, *friendship_graph* and *business_table*. Using these we read the csv-files using the appropriate delimiters.

2.6 Task 6

a - Inner join review table and business table on business_id column

Here we use the DataFrames from Task 5. We create temporary views called `businesses` and `reviews`. These two tables are joined using the query

```
SELECT * FROM businesses, reviews
WHERE businesses.business_id = reviews.business_id.
```

b - Save the new table in a temporary table

Using the result from Task 6a) we store the new table in a temporary table using the `.createOrReplaceTempTable()` function on the DataFrame.

c - Find the number of reviews for each user in the review table for top 20 users with the most number of reviews sorted descendingly

To solve this task we create a DataFrame just as in task a) and b). From this dataframe we create a TempView `reviews`, and we issue the query:

```
SELECT user_id, COUNT(review_id) AS review_count
FROM reviews
GROUP BY user_id
ORDER BY review_count DESC
LIMIT 20
```